

FILE COPY

2

AD-A225 125

ANNUAL REPORT
VOLUME 3
TASK 3: SPECIAL STUDIES

REPORT NO. AR-0142-90-001

July 23, 1990

DTIC
ELECTE
AUG 03 1990
S
D
Cg

GUIDANCE, NAVIGATION AND CONTROL
DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Contract Data Requirements List Item A005

Period Covered: FY 90

Type Report: Annual

~~SECRET~~

**ANNUAL REPORT
VOLUME 3
TASK 3: SPECIAL STUDIES**

July 23, 1990

Authors

Cecil O. Alford and Philip R. Bingham

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Eugene L. Sanders

USASDC

Contract Monitor

Cecil O. Alford

Georgia Tech

Project Director

Copyright 1990

Georgia Tech Research Corporation

Centennial Research Building

Atlanta, Georgia 30332

DISCLAIMER

DISCLAIMER STATEMENT - The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

DISTRIBUTION CONTROL

- (1) **DISTRIBUTION STATEMENT** - Approved for public release; distribution is unlimited.
- (2) This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227 - 7013, October 1988.

Accession No.	
NTIS	100-41
Dist	1Ae
Unannounced	
Justification	
By	
Distribution	
Approved for	
Dist	A-1



TASK 3: SPECIAL STUDIES

1. INTRODUCTION

Task 3 was set up with two objectives. The first was to establish interfaces between Georgia Tech and other programs such as LATS, LETS, AHAT and KDEC. Each of these programs requires specific interactions and data exchange with Georgia Tech. Some of these will eventually require an interface specification to define hardware and software boundaries between Georgia Tech and the other programs.

The second objective of Task 3 is to resolve issues which were not anticipated in the original contract, but which need resolution for the contract to move forward. These issues are sometimes quite difficult and beyond the resources of the contract. However, each issue is at least defined and evaluated in importance. If further work is justified it is left open for USASDC assignment.

/

C

2. CONTRACT INTERFACES

Interfaces have been established with all four programs which merge with Georgia Tech. These are AHAT, LATS, LETS and KDEC. A point of contact has been established at Georgia Tech for each program. A summary of each follows.

2.1 AHAT

Dr. W.S. Tan is the point of contact for VLSI design and GN&C Processor issues. Andy Register is the Georgia Tech interface for all AHAT meetings. This interface is working smoothly. Georgia Tech VLSI design tapes are being delivered to Harris on a regular basis. Questions that arise are resolved quickly if the information is at Georgia Tech. The specific interface between the AHAT processor and the Georgia Tech test facility has not been addressed. Two issues are outstanding. The first is a programming port for instruction loading and editing. The second is a data port for FPA pixel information. These two have not been defined to a level suitable for Georgia Tech to begin an interface design. At some point these interfaces will be defined, a specification will be written and Georgia Tech will begin development of suitable test devices.

2.2 LATS

The LATS facility will produce FPA pixel data and process it to some level. The data will be sent from LATS to a PFP over a SCSI link. This data path has been specified to a degree. The data rates and some of the data elements are known. Andy Henshaw serves as the interface for Georgia Tech. He has talked with LMSC (PARC) and is designing a specific interface for the PFP, to receive data over a SCSI port, cluster and centroid the data, and transit the centroids to the EXOSIM simulation. The interface is sufficient for 32 x 32 FPAs at frame rates of 100 hz.

A second SCSI port is also provided to transmit data to LATS. This would most likely be control information from the GN&C Processor. The exact data and data rates have not been specified. However, one SCSI port should be sufficient for this command/control channel.

2.3 LETS

Andy Henshaw is the primary interface between Georgia Tech and LETS. His interface is a SCSI port to transmit data from the PFP to the LETS VAX. Randy Abler is a secondary interface for LETS. His responsibility is an ETHERNET interfaces between the SUN PFP host and the LETS ETHERNET. These interfaces present no difficulties and can be met with standard products.

2.4 KDEC

Georgia Tech has provided PFP programming classes to support KDEC. Two manuals have been written to assist users in programming the PFP. EXOSIM is being

converted to a parallel format for execution on the PFP. When KDEC has a working PFP, interface issues on EXOSIM will be directed to Richard Pitts.

2.5 Other

Interfaces with Draper Laboratories, LEAP, GBI, and KHILS are not being developed. As these entities become important and require special attention, new interfaces will be developed.

2.6 Working Groups

To support the interfaces and to ensure that all participants have a forum for presenting requirements, specifications and problems, a set of four working groups (WG) has been established. Each WG is directed at a specific technology area. The four groups and their leaders are:

- (a) Parallel Simulation Technology Working Group - Tom Collins (404)894-2509. This group will focus on parallel simulation techniques and tools, simulation languages, and parallel simulations such as EXOSIM and LEAP.
- (b) Seeker/Scene Emulation Technology Working Group - Andrew Henshaw (404)894-2521. This group will focus on scene generation techniques, seeker modeling, benchmark test and evaluation data for GN&C processors, and emulation accuracy assessment.
- (c) GN&C Processor Working Group - Andy Register (404)894-3812. This group will focus on processor architectures, VLSI design tools, VLSI fabrication, packaging and benchmarks for processor evaluation and testing.
- (d) Parallel Ada Software Working Group - Dr. W.S.. Tan (404) 894-2508. This group will focus on development tools for embedded Ada software, parallel implementation techniques, test and evaluation of embedded Ada software, and benchmark test programs for parallel simulation and GN&C processors.

3. TECHNICAL ISSUES

3.1 Delayed Gamma Model

The rejection of gamma spikes in the signal processor presents a significant problem. Data rates are required to be quite high (10 khz) which requires a large number of A/D converters. Processing, although quite simple, requires a large number of chips to support the high data rate. An effort was made to develop a reasonable model to test various gamma rejection methods. This work is reported in Volume 5. Ultimately an analog scheme is preferred since it would eliminate chips and reduce the package size and weight. It is too early to tell whether analog techniques are sufficient, but the assumption is being made that they will eventually do the job. In case they do not, a digital plan is available and can be implemented.

One of the outputs from the investigation was a testing procedure. Georgia Tech has a preliminary design for a gamma injection circuit which could be used to test signal processing hardware with digital gamma rejection. Since the input is known, it is possible to get an exact measure on any gamma rejection hardware. This could prove useful, but only for digital methods, since analog schemes require a different injection technique. Nothing further will be pursued in the gamma rejection or testing area unless the analog technique fails.

3.2 Parallel EXOSIM

A parallel simulation for an EXO-Atmospheric KEW interceptor has been developed by Coleman Research Corporation. Converting the Fortran code for PFP execution has been a primary objective of Task 1. The unusual problems encountered by the Task 1 group and by Dynetics Inc., necessitated additional effort and approaches. Section 4 is a summary of a conversion methodology developed under Task 3. Section 5 applies the methodology to EXOSIM and discusses the parallel implementation.

4. PARALLEL PROGRAMMING METHODOLOGY

4.1 Introduction

A parallel programming methodology has been developed based on data flow graphs, and the definition of a primitive language which permits a data flow graph to be translated into a computational structure. These concepts are discussed in the following sections and applied to Parallel EXOSIM in Section 5.

4.2 Data Flow Graphs

The objective is to re-create the engineering block diagram from the Fortran code. Obviously, the original block diagram would be best, but is not available in this case, just as it isn't for many "Dusty Deck" Fortran programs. However, the code is broken up into small subroutine segments which is a tremendous aid in recovering the engineering structure.

The variables which are passed to a Partition are identified and placed on the left side of the data flow graph. Variables which are passed out of a Partition are identified and placed on the right side of the flow graph. Computational data flow and time proceeds from left to right in this graph. As each subroutine call is encountered, it is entered as a block (to be expanded later) with the input variables on the left and output variables on the right. In order to construct the graph some operational primitives must be defined.

4.3. Execution Primitives

4.3.1 Input/Output Primitives

Input and Output Primitives are shown in Table 4.1. Each data flow graph has input primitives on the left and output primitives on the right. The symbols in Table 4.1 identify the variable by name, type and bit width. The types are floating point 32-bit (R) or 64-bit (D), integer 16-bit (I) or 32-bit (L) and logical which can have 1 bit up to N bits where N is a specified parameter. In addition some variables may not fall into any of these types so a special type (S) is used to identify these unusual cases.

4.3.2 Scalar Primitives

Operations consist of addition, subtraction, multiplication, division, functions, tables of one variable and tables of two variables. These are defined in Table 4.2. Other primitives can be added such as Tables of three variables, integration, and specific nonlinear functions. For example, limiting and deadzone are often given as distinct primitives and not included within the one variable TAB() function. For a coarse grain computational view this detail is not needed. For further expansion it can be useful. Note that Table 4.2 is identified as scalar. Since these operations occur so often it is best to be able to isolate them as a distinct group.

2

4.1

Table X. Input and Output Variable Definitions & Symbols




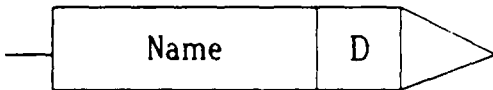

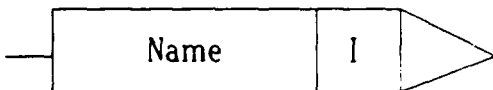

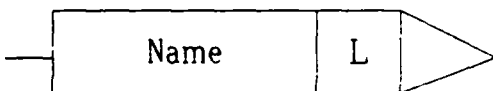
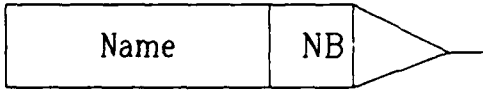
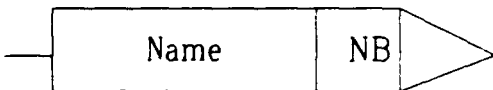

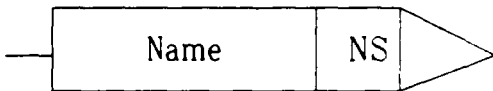
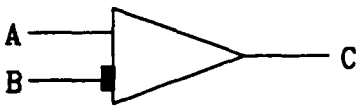
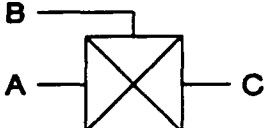
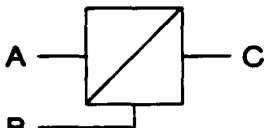
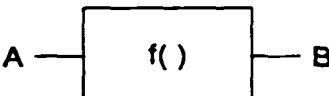
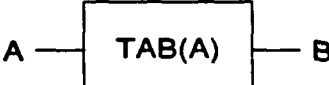
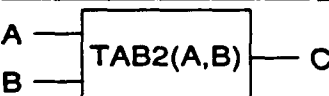
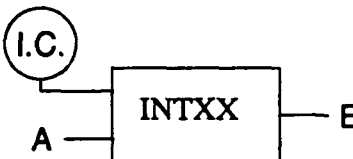
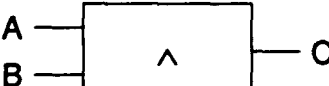
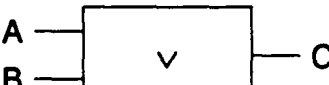
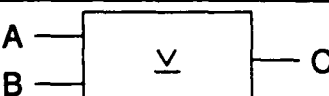
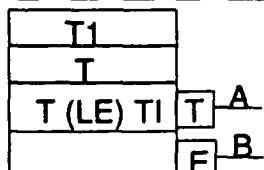
DEFINITION	INPUT SYMBOL	OUTPUT SYMBOL
32-bit Floating Point		
64-bit Floating Point		
16-bit Integer		
32-bit Integer		
N-bit Boolean (N is an integer)		
N-bit Special Floating Point or Integer (N is an integer)		

Table 4. Scalar Execution Primitives

10

PRIMITIVE	SYMBOL	ACTION
ADD/SUB		$C = A - B$
MULTIPLY		$C = A * B$
DIVIDE		$C = A / B$
FUNCTION		$B = f(A)$ $f = \sin, \cos, \tan, \text{sqrt}$ $\arcsin, \arccos, \arctan$ $\exp, \log, \ln, \text{abs}$
TABLE ONE VARIABLE		$B = \text{TAB}(A)$
TABLE TWO VARIABLES		$C = \text{TAB2}(A, B)$
INTTEGRATION		$B = \int A(t) dt + IC..$ $\text{INTXX} = \text{INTEU (EULER)}$ $\text{INTTR (TRAPEZOIDAL)}$ $\text{INTRK (RUNGE KUTTA)}$
AND		$C = A \wedge B$
OR		$C = A \vee B$
EXCLUSIVE OR		$C = A \underline{\vee} B$
DECISION		IF (T LE T1) Then A Else B LE is A Logical Expression

Note that the summation primitive has a heavy black rectangle on the B input. This is used to indicate a signal is negated. If the symbol is omitted $C = A + B$. The symbol can be used on any input signal line for arithmetic variables. It is not defined for logical variables (though it could be defined as ones complement). The Decision Primitive is explained in Section 4.3.4.

Conversion depends on the representation. For IEEE floating point conversion, the exponent is 8-bits for F (11-bits for D) and the mantissa is 24-bits for F (53-bits for D). Conversion from F to D uses a zero fill. Conversion from D to F uses truncation or rounding on the mantissa.

Integers can be converted exactly by putting the integer in fractional form with an exponent. The integer is already in binary format therefore a shift of the binary point determines the exponent. A 16-bit integer will convert unchanged into F or D format. A 32-bit integer will be truncated or rounded for F and unchanged for D.

In converting F or D to I or L the F or D number is put in expanded form and the integer part selected. Truncation or rounding to the nearest integer is used in the conversion.

Selection of truncation or rounding is a hardware option that is embedded in the compiler. The user normally does not have a choice. Conversion Primitives are shown in Table 4.3.

4.3.3 Vector Primitives

A set of vector primitives is given in Table 4.4 These are based on the definitions:

\overline{x}_k = Vector with k Elements

$x(1)$ = Scalar Value of \overline{x}_k

a, b, c = Scalars

$a_{m \times n}$ = Matrix with m Rows and n Columns

TRAN = Transpose of a Matrix or Vector

DOT = Vector Dot Product

CROSS = Vector Cross Product


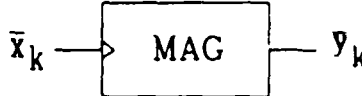



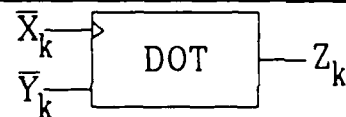
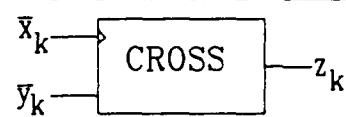
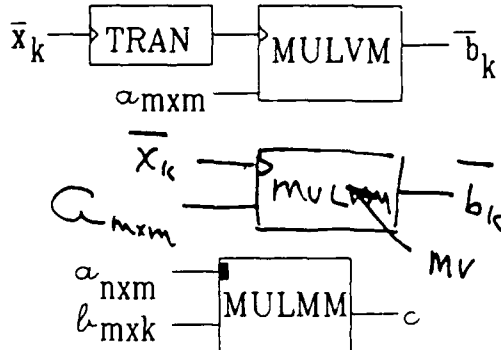
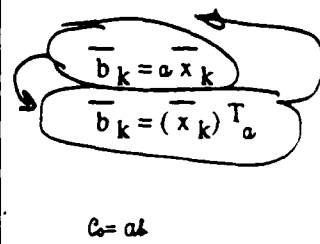
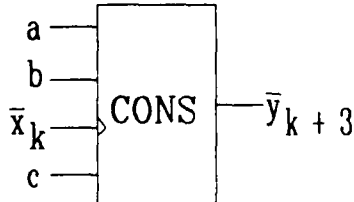
DET = Determinant of a Matrix

MAX = Maximum Value of a Matrix or Vector

4.3
12
Table Y. Data Type Conversion Primitives

DEFINITION	SYMBOL	ACTION
Convert F TO D	CONFD	Convert 32-bit floating point number to 64-bit floating point number
Convert D To F	CONDF	Convert 64-bit floating point number to 32-bit floating point number
Convert I TO L	CONIL	Convert 16-bit integer to 32-bit integer
Convert L To I	CONLI	Convert 32-bit integer to 16-bit integer
Convert I(L) To F	CONI(L)F	Convert 16-bit integer (32-bit integer) to 32-bit floating point number
Convert I(L) To D	CONI(L)D	Convert 16-bit (32-bit integer) to 64-bit floating point number
Convert F(D) To I	CONF(D)I	Convert 32-bit (64-bit) floating point number to 16-bit integer
Convert F(D) To L	CONF(D)L	Convert 32-bit (64-bit) floating point number to 32-bit integer

Table 5. Vector Execution Primitives

PRIMITIVE	SYMBOL	OPERATION
SCALE		$\bar{y}_k = a \bar{x}_k$
MAGNITUDE		$\bar{y}_k = \text{sqrt} \sum_{i=0}^k x(i)^2$
MAXIMUM		$a = \text{MAX}(x(1), x(2), \dots, x(k))$
MINIMUM		$a = \text{MIN}(x(1), x(2), \dots, x(k))$
TRANSPOSE		$\bar{y}_k = (\bar{x}_k)^T$
DOT PRODUCT		$\bar{z}_k = (\bar{x}_k)^T * (\bar{y}_k)$
CROSS PRODUCT		$\bar{z}_k = \bar{x}_k \times \bar{y}_k$
MULTIPLY		
CONSTRUCT VECTOR		$\bar{y}_{k+3} = \begin{bmatrix} a \\ b \\ \bar{x}_k \\ c \end{bmatrix}$

MIN	=	Minimum Value of a Matrix or Vector
MAG	=	Magnitude of a Vector
UNIT	=	Convert Vector to a Unit Vector
SCALE	=	Scale a Vector or Matrix by a Scalar
CONV	=	Construct a Vector from Scalar and Vector Parts

These vector primitives permit the code to be reduced to an extremely compact form. This makes the coarse grain parallelism much more obvious. the specific implementation of these vector operations is not disclosed. There are many possible implementations which take advantage of the inherent parallelism, but this will be embedded until a fine grain structure is necessary. The next step is to convert the code, using the primitives, to a concise data flow representation.

4.3.4 Decision Primitive

The Decision Primitive is represented by the symbol shown in Figure 4.1. The symbol graphically represents an If-Then-Else program control statement in a data flow graph. The conditions are

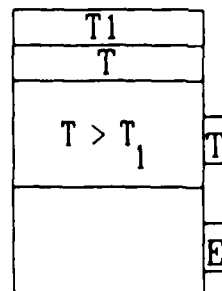


Figure 4.1. Decision Primitive Symbol

given in the top part of the symbol with the decision variables shown on top. If these are changed by other statements, the data flow graph should indicate this with a data line to T1 or T. If these are constants they should not be shown above the symbol. The "T" branch is the "Then" exit while the "E" branch is the "Else" exit.

An example of a complex IF-Then-Else structure is shown in Figure 4.2. This has been converted to the data flow graph shown in Figure 4.3.

A second example is shown in Figure 4.4. This example represents a nested conditional statement and is implemented as shown in Figure 4.5. An alternate

implementation is to use an extended primitive as shown in Figure 4.6. This gives the same information and either can be used. The extended Decision Primitive represents the program statement If-Then-Else If-Else If- . . .-Else.

The advantage of the graph is clarity in presenting conditions, output and exactly where the paths diverge and converge.

4.3.5 Constant Primitive

Constants play a unique role in a data flow graph. they can have names and type, similar to variables but are not to be treated as variables. In a program, constants often appear as variables and can only be sorted out by referring to an input data list. To avoid placing constants on the left of a data flow graph and treating them as variables it is best to use the letter C as a prefix to identify all constants. The second letter defines the type using the letters F,D,I,L,B, and S which have the same definitions as in Section 4.3.1. The symbols are defined in Table 4.5.

On a data flow graph all constants are listed in a table. An example table is shown as Table 4.6. The Constant Definition Table gives all constants by name, which includes type. The values are given in column 2. Any additional information is given in column 3. This column would explain constants of the CB or CS type.

By identifying constants in this way, the task of data dependency is made easier. Variables are automatically sorted out and constants are not confused as variables which must be computed.

4.3.6 Delay Primitive

In simulation programming the delay operator is crucially important. This operator is defined by the symbol and operation shown in Figure 4.7. The output b is not defined at $k=0$ unless the initial condition $IC = b(0)$ is specified. Thus every delay operator must have an associated I.C. In a computer program this

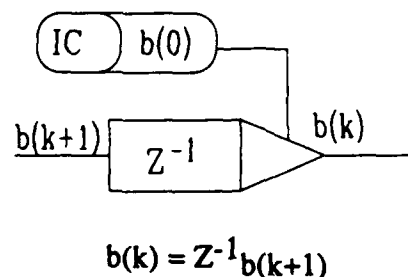


Figure 4.7 Unit Delay Operator

```
IF (T > T1) Then
  A
  IF (T>T3) Then
    E
  Else IF (T>T4) Then
    F
  Else IF (T>T5)Then
    G
  Else
    H
  End IF
Else
  B
  IF(T>T2) Then
    C
  Else
    D
  End IF
End IF
```

Figure 4.2. Complex Decision Logic

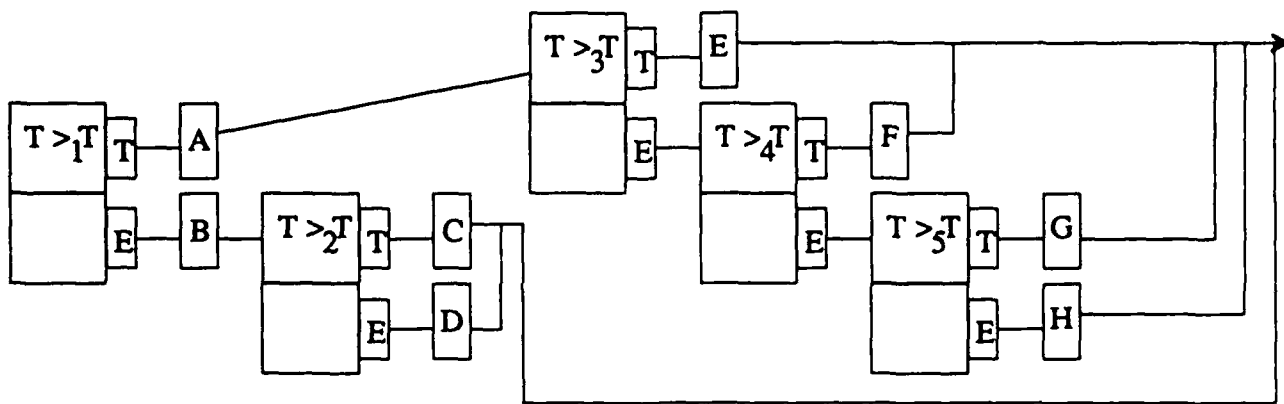


Figure 3. Primitive Implementation of Figure 2.

x.1

x.2

```
IF (T > T1) Then
    A
Else IF (T > T2) Then
    B
Else IF (T > T3) Then
    C
Else IF (T > T4) Then
    D
Else
    E
End IF
```

Figure 4.4. Decision Tree

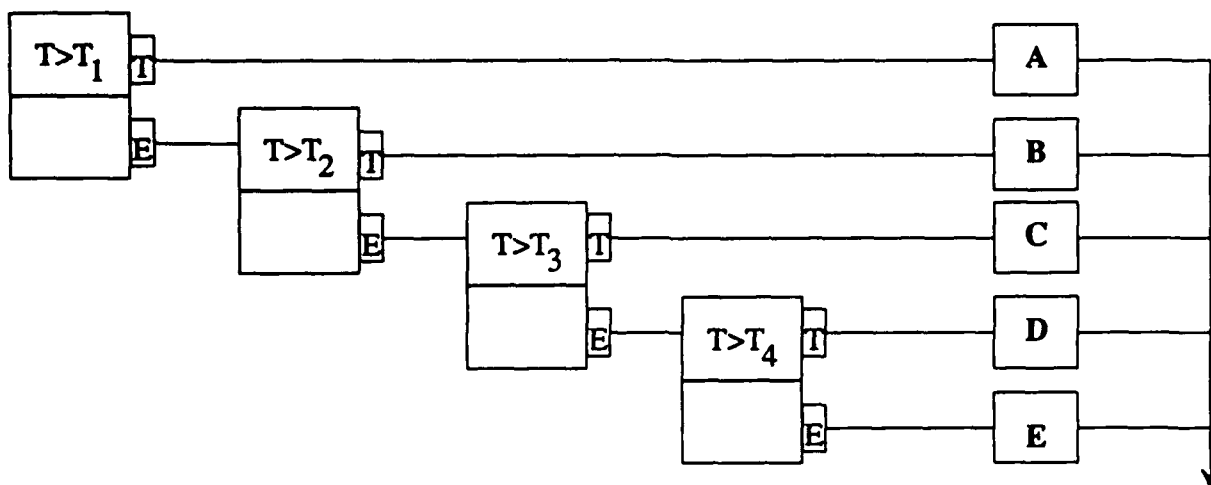


Figure 4.5. Primitive Implementation of Figure 4.4

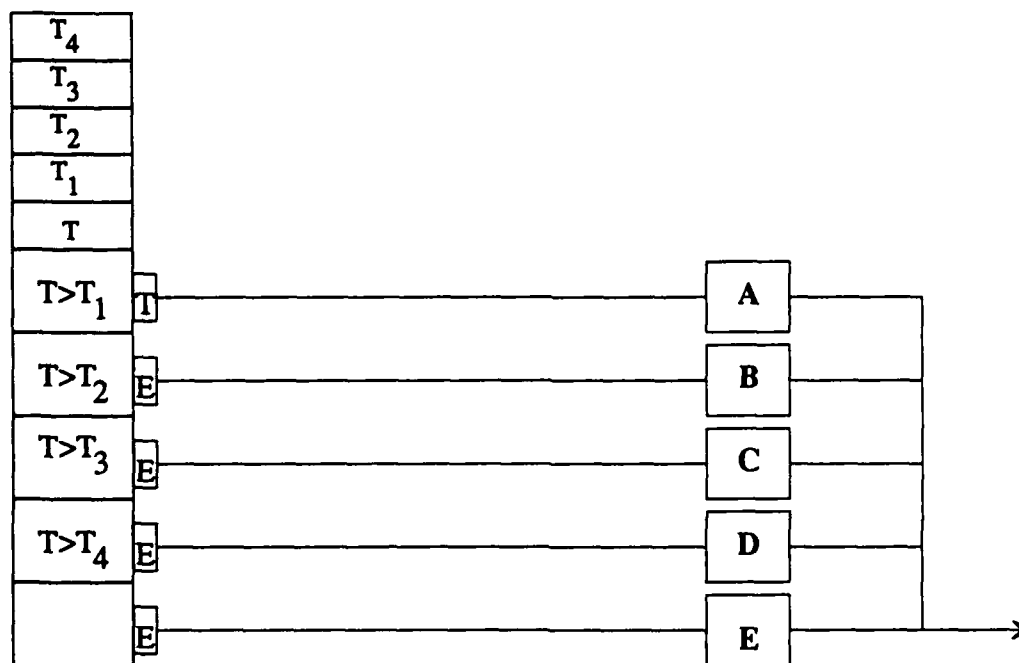


Figure 4.6. Extended Primitive Implementation of Figure 4.4

Table 4.5. Constant Primitive

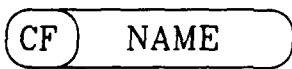
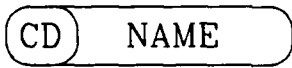


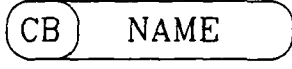

DEFINITION	SYMBOL	VALUE
Floating Point Constant		32-bit Floating Point Number
Floating Point Constant		64-bit Floating Point Number
Integer Constant		16-bit Integer
Long Integer Constant		32-bit Integer
Logical Constant		N-bit Logical Constant
Special Constant		N-bit Special Type Constant

Table 4.6. Constant Definition Table

NAME (1)	VALUE (2)	INFORMATION (3)

- (1) Name is taken from the flow graph.
- (2) Value is the current simulation number
- (3) Special information relates number of bits and other data for CB and CS type constants.

shows up in code as $\text{Var1}(k+1) = \text{Var1}(k) + f(\text{Var1}(k), \text{Var2}(k))$ where k refers to the previous time step and $(k+1)$ the current time step. This can be implemented as shown in Figure 4.8

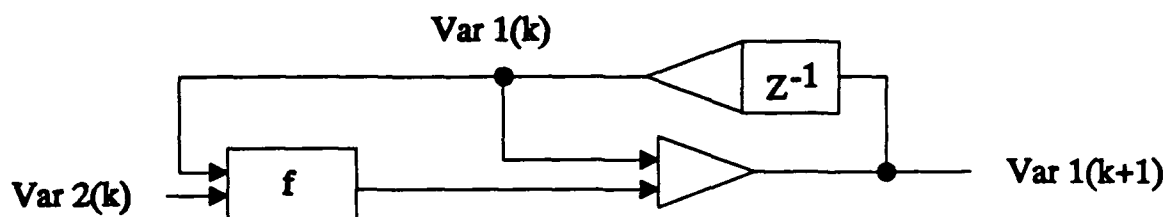


Figure 4.8 Implementation of Delayed Variable Calculation

In order to compute $\text{Var1}(k+1)$ it is necessary to have an initial condition $\text{Var1}(0)$ and $\text{Var2}(0)$. $\text{Var2}(k)$ is generated in a similar manner by another equation. There are two ways this can happen as shown in Figure 4.9

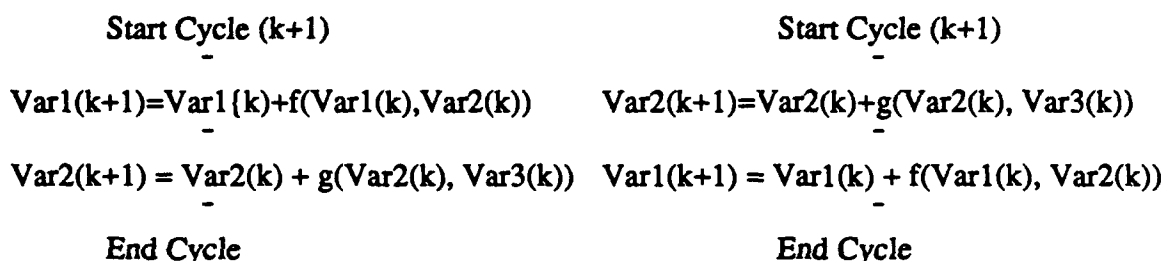


Figure 4.9 Programming Sequence For Cycle (k+1)

Both versions shown in Figure 4.9 are implemented as shown in Figure 4.10. Because all $(k+1)$ values are dependent on (k) values and no $(k+1)$ values, the two functions $\text{Var1}(k+1)$ and $\text{Var2}(k+1)$ can be solved independently and in parallel.

Many programs are written without regard to the cycle number. If this is done, all the (k) and $(k+1)$ values are removed from Figure 4.9. The implementation for the two codes is then different. The version on the left in Figure 3 (Var1 precedes Var2) is implemented exactly as shown in Figure 4.10. The version on the right in Figure 3 (Var2 precedes Var1) is implemented as shown in Figure 4.11 (with appropriate assumptions on Var3). In Figure 4.11 the two functions become dependent and must be computed sequentially; Var3 followed by Var2 followed by Var1 .

In extracting parallelism within a sequential program it is necessary to distinguish between these two code variations. If "time stamps" are available as in Figure 4.9, it is relatively easy to trace the paths which produce $\text{Var1}(k+1)$ and $\text{Var2}(k+1)$. This would then lead to a data flow graph as shown in Figure 4.10 or Figure 4.11. In order to do this it is helpful to convert the code to a "time stamped" format. This can be done by entering the code at the beginning of cycle $k+1$. All variables on the left hand side of the replacement sign will become $\text{Var}(k+1)$. Variables on the right hand side of the replacement operator will become $\text{Var}(k)$ and in some cases $\text{Var}(k+1)$. Whenever a

variable on the R.H.S. appears after a similar variable on the L.H.S. it becomes $\text{Var}(k+1)$. If the variable appears on the R.H.S. before the same variable appears on the L.H.S. it will be $\text{Var}(k)$. When in the same line as the variable on the L.H.S., the variable on the R.H.S. is $\text{Var}(k)$.

Calls to subroutines and functions must also be observed carefully. Variables passed to the subroutine will be $\text{Var}(k+1)$ if the variable has appeared on the L.H.S prior to the calling statement. Otherwise the variable will be $\text{Var}(k)$. The returned variables are analyzed within the subroutine in the same manner as the primary program. However, once they are "time stamped" it becomes easier to couple them into succeeding routines.

Constants are another item which need identification. If not identified as such, the data flow diagram treats them as variables. This increases the difficulty of the search procedure and also requires "time stamping". It is essential to use "C" as a prefix for constants. All labels related to constants are then readily identifiable and the data flow graph is much clearer. Also the list of variables would be reduced to true variables.

Initial conditions are identified by the presence of delay blocks in the data flow graph. Every variable which is an input to a delay block requires an initial condition. The delay blocks can be used to generate an appropriate list of variables and their initial conditions. This is one of the areas most subject to error. Many compilers will initialize variables to zero, but others do not. Good simulation practice requires a list of variables which requires initial conditions and their respective values.

After the sequential code is converted using these techniques, each major section can be partitioned with input and output variables. In this sense a section becomes a subroutine call, though it is not written as such. The output variables for each section can be determined from a data flow graph which traces the computation back to the set of input variables. The "time stamp" on all required variables indicates immediately those variables which are independent and those which must wait on other computations. There are three cases which are illustrated in Figure 4.12. In Figure 4.12(a) computations for $V4(k)$ can begin at the beginning of cycle $(k+1)$. This block only needs to receive $V1(k)$, $V2(k)$ and $V3(k)$ for computation to begin. These values will be output during cycle (k) .

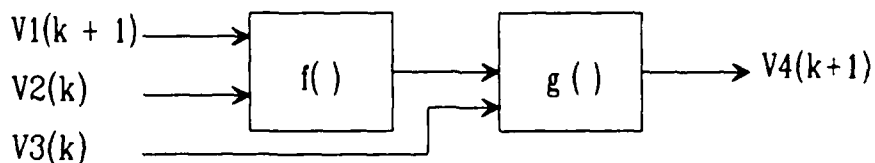


Figure 4.12(a) Completely Independent Block

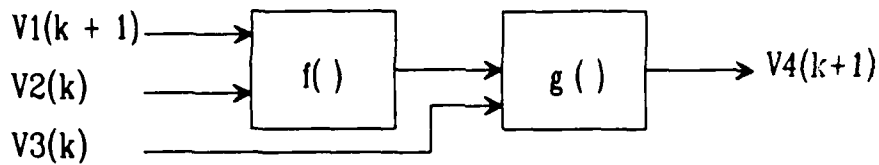


Figure 4.12(b) Completely Dependent Block

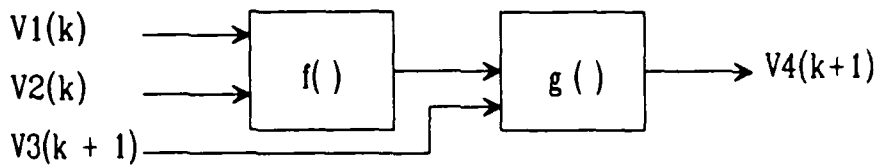


Figure 4.12(c) Partially Dependent Block

Figure 4.12. Data Flow Dependency Conditions

In Figure 4.12(b) the variable $V1(k+1)$ must be computed before the function $f()$ can start. Hence the entire computation for $V4(k+1)$ must wait for $V1(k+1)$. After $V1(k+1)$ is calculated all of the computations for $V4(k+1)$ can proceed. If the processor calculating $V1(k+1)$ has no additional duties, the computation of $V4(k+1)$ can be assigned to that processor. Otherwise, the value $V1(k+1)$ can be transferred to another processor and $V4(k+1)$ is computed in parallel with other functions.

Figure 4.12(c) illustrates a partially dependent graph. The function $f()$ can begin at the start of cycle $(k+1)$. The output from $f()$ and $V3(k+1)$ are needed for $g()$. When $V3(k+1)$ is completed, it can be transferred to the processor computing $g()$. When the output of $f()$ and $V3(k+1)$ are both available the computation of $g()$ proceeds. This permits parallel processing wherein $f()$ is computed in parallel with $V3(k+1)$. The function $g()$ may also be computed in parallel with other blocks.

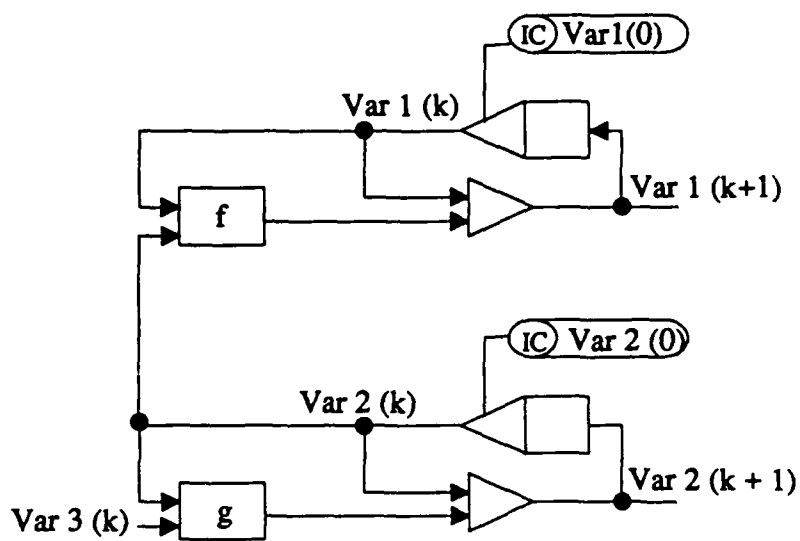


Figure 4.10. Implementation of Figure 3 (left)

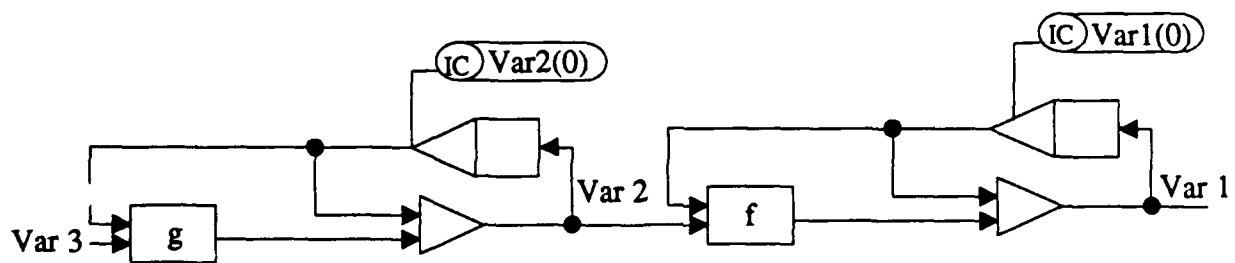


Figure 4.11. Implementation of Figure 3 (right)

There are three cases which are illustrated in Figure 4.12. In Figure 4.12(a) computations for $V4(k)$ can begin at the beginning of cycle $(k+1)$. This block only needs to receive $V1(k)$, $V2(k)$ and $V3(k)$ for computation to begin. These values will be output during cycle (k) .

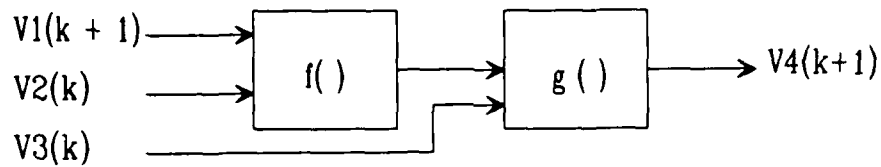


Figure 4.12(a) Completely Independent Block

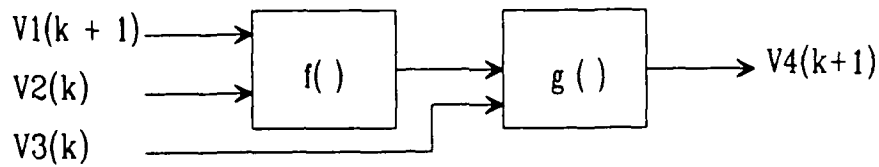


Figure 4.12(b) Completely Dependent Block

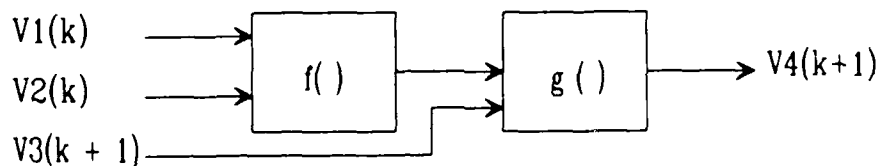


Figure 4.12(c) Partially Dependent Block

Figure 4.12. Data Flow Dependency Conditions

5. PARALLEL EXOSIM SIMULATION

5.1 Introduction

An Exoatmospheric Kinetic Energy Weapons Interceptor Simulation was initiated by BDM Corp. and completed by Coleman Research Corporation. The final report, EXOSIM V2.0, was delivered to the U.S. Army Strategic Defense Command, October 15, 1989 under contract number _____. This simulation, written in FORTRAN, was developed to run on a serial computer such as a VAX. The objective at Georgia Tech is to convert the simulation to Ada and to restructure it in a parallel format for execution on the Georgia Tech Parallel Function Processor. A second objective is to preserve the functional nature of the simulation so that changes can be easily incorporated into the executing code.

An analysis of the serial code revealed several problem areas which make the conversion more difficult. These are:

1. Double Precision Variables.
2. Dependent Partitions.
3. Event driven code.
4. System States difficult to identify.
5. Inadequate descriptions of system models.

Two approaches were selected for the conversion. The first, and most direct, was to examine the code, extract the physics, and re-write the simulation. This work is underway. The second approach was initiated by Dynetics Inc. Dynetics removed the event driven structure, produced an "equivalent unclassified version", and identified partitions which could be run in parallel. Dynetics worked with EXOSIM v1.0 since v2.0 was not available when they began work. They were only able to complete the Boost phase before funds were exhausted. This code was delivered to Georgia Tech and is known as EXOSIM Boost2.

Boost 2 incorporates virtually all of the EXOSIM v1.0 code. Extending Boost 2 to include the KV only requires the addition of the KV autopilot, ACS thruster, VCS thruster and the seeker code. With the exception of the seeker code, these new components are similar to other modules which are used for the boost phase. Georgia Tech has worked with Boost 2, developing conversion techniques and a methodology. Once Boost 2 is converted, the KV phase will be added. The remainder of this section discusses the methodology developed for the Boost 2 conversion effort.

The Boost 2 code consists of five partitions which have been designed to operate in parallel. Parallel operation was verified by running the partitions on a serial computer in various sequences. The sequence 1,2,3,4,5 is considered the baseline. Other variations

in the order produces certain errors in the code and generates slight errors in the ALT (altitude) variable. If the error was under 2000 feet out of 229,830 feet it was considered reasonable. Errors are caused because some variables are delayed by one cycle from their order when executed in the baseline sequence. This causes slight variations in these variables which ripple through all variables.

5.2 Timing Analysis

In EXOSIM there are five partitions for the BOOST2 program. This code flies the KEW vehicle from launch through boost phase 1 to the end of boost phase 2. Most of the subroutines are active during both phases, but a number are only active in one flight phase.

A timing routine was written to determine execution delay for each partition. The values were accumulated over 1000 simulation clock periods ($T_s = 0.001$ seconds).

From these timing results maximum and minimum values for each partition, for each flight phase, were selected and are listed in Table 5.1. Two things are obvious from the results; (1) the difference between the maximum and minimum for a given partition is relatively small, and (2) the differences between the maxima across the partitions is quite large. This shows that the code within a partition does not vary very much (although the averaging smoothed the results) but the computational balance is poor. Hence, Table 5.1 says to focus on Partition 1.

The timing program was used again within Partition 1 to assess the distribution of compute time by subroutines. The results are shown in Table 5.2. This Table indicates ATMOS and AERO as the subroutines which require the most computation. However, rather than attacking these two subroutines, which could be done, with positive results, it is best to look at Partition 1 as a whole and assess other possibilities at the same time. For example, as written, Partition 1 has a total of _____ double precision variables which must be transferred into or out of the processor each cycle. The transfer time for these variables is an enormous problem. Therefore, any parallel partition should address variable transfer as well as compute time.

Note: All timing information was taken from the Intel iSBC 286/12 processor. Values for the iSBC 386/12 would be reduced by a factor of 4 (approximately). The iSBC 486/12 would reduce the values by a factor of 16 (approximately).

5.3 Code Conversion

Figure 5.1 shows the result of converting the Partition 1 code into a data flow graph. Variables are transferred in on the left at time t_n . Computation proceeds in a left to right flow with variables generated for time t_{n+1} . These are transferred between the partitions and the cycle is repeated.

The data flow graph makes obvious the enormous amount of parallelism which is present. Instead of one processor, ten or more could be used to start the computation for Partition 1. All that is necessary are the variables shown to begin computation along one of the input paths. The availability of these variables depends on the sending partition and will be addressed later.

The second feature of the graph is data flow dependence which sets the longest chain in the graph. At this point it can't be concluded the longest chain is the longest in time, since blocks are not equal in time delay, but it does give an idea of how the analysis should proceed. Furthermore, the timing analysis has already shown that ATMOS and AERO are dominant time delays in Partition 1 and both are in the longest chain. Hence, attention must now turn to these two subroutines.

A few additional comments are needed on the main partition. In the original version all variables were declared as double precision. This is an exorbitant waste of computing effort and in many cases fails to model the system blocks appropriately. In Partition 1 the only case for double precision is the generation of position variables XYZ and variations thereof. All other variables should be handled as 32-bit floating point values or integers. This accuracy is more than sufficient for ATMOS, AERO, INTEG (except for XYZ)etc. To test the system sensitivity, all partition variables except XYZ and its variation were changed to single precision. The output values were compared to those obtained for the double precision case. The single precision values were subtracted from the double precision values to create an error file. The errors are given in Table 6.

5.4 Conversion of Subroutines ATMOS and AERO

The two subroutines which require significant amounts of compute time are ATMOS and AERO. ATMOS is used to generate atmospheric values as a function of altitude. The subroutine, shown as a flow graph in Figure 5.2, exhibits six table look-ups which can all be executed in parallel. This in itself represents a 6x speedup over the previous method. By using single precision instead of double precision, an additional 2x can be gained giving an expected 12x performance gain for this segment. In order to do this, the appropriate variables must be routed to the processors containing the tables. The table look-up function can be implemented in a variety of ways (as discussed in Section 5.5). indicated in the flow graph, but it is safe to assume it will execute faster than the original. If the variables are sent from the appropriate processors, a table look-up processor is only required to execute the following pseudo code;

```
Receive (ALT , 2)

Receive (MACH ,2)

Compute (      )

Send (PRESS,2)
```

5.1
Table 1. Max - Min Partition Timing Values

Boost Phase 1		
Partition	Maximum	Minimum
1		
2		
3		
4		
5		
Boost Phase 2		
1		
2		
3		
4		
5		

5.2

1

Table 2. Max - Min Partition Timing 1 Values

Boost Phase 1		
Partition	Maximum	Minimum
1		
2		
3		
4		
5		
6		
7		
8		
Boost Phase 2		
1		
2		
3		
4		
5		
6		
7		
8		

Figure 5.1 Data Flow Representation of Partition 1

In the Receive statement a variable such as ALT is removed from the input buffer. The integer 2 indicates the variable consists of two, 16-bit parts. It has to be known the two parts constitute a 32-bit floating point number and not a 32-bit integer. The variable ALT will be assembled as a floating point number for computing the required output. The compute statement represents a procedure to obtain the output, given the input values. The Send statement, outputs the variable PRESS to the output buffer. Since the variable is a 32-bit floating point number it is output as two, 16-bit parts. Each part will be transmitted over the crossbar to one or more destination processors.

5.5 Tables

Look-up tables are used in EXOSIM for atmospheric properties (in ATMOS), missile properties (in MASSPR) and aerodynamic properties (in AERO). Most tables are functions of a single variable although three (CN, CA, XCP) are functions of two variables. Sample plots of look-up tables from BOOST2 are given in Appendix A.

A nonlinear function of one variable is implemented in a table using non-uniform spacing on the x-axis. Linear interpolation is used to obtain the value $F(x)$ by

$$\text{Del } x = \frac{x - x_k}{x_{k+1} - x_k} \quad (5.1)$$

$$F(x) = (\text{Del } x (F(x_{k+1}) - F(x_k))) \quad (5.2)$$

A linear search or a binary search is used to locate the values x_{k+1} and x_k .

A much more efficient method is to use more points (requiring more memory) and divide the x-axis into segments with uniform spacing. For example the range $x = [0, 100]$ could be covered by $[0, 1, 0.1]$, $[1, 10, 1]$, $[10, 100, 10]$. Each of these triplets gives the range of x and the delta x . To compute the function e^{-x} on $[0, 100]$ the function values are shown in Table 5.3. Two methods can then be used to obtain $F(x)$.

5.5.1 Direct Look-Up

If the table is sufficiently dense there is little to be gained by linear interpolation. Assuming this is sufficient the algorithm is shown in Figure 5.3:

Figure 5.2 ATMOS Subroutine Flow Graph

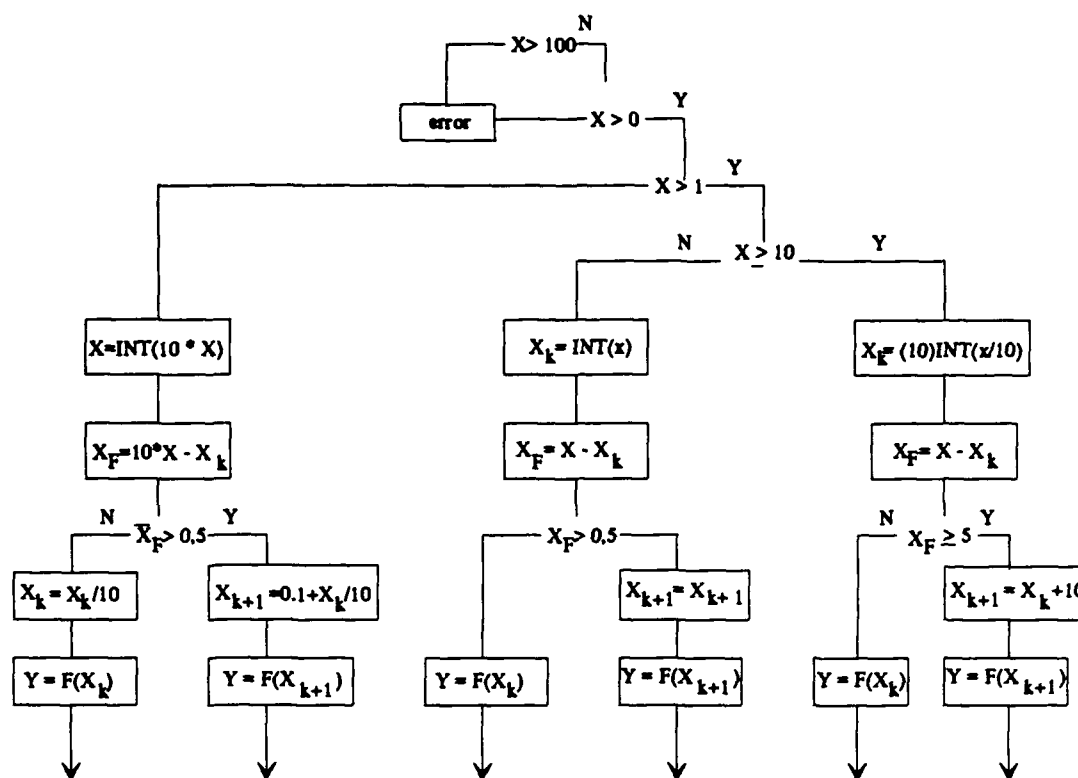


Figure 5.3 Flow Graph For Table Look-UP Function

For the exponential function in Table 5.3, the errors are largest over $[0,0.1]$, $[1,2]$, and $[10, 20]$. The maximum error in each case is,

$$\text{err}_{\max} = e^{-0} - e^{-0.05} = 1 - 0.9512294 = .04877$$

$$\text{err}_{\max} = e^{-1} - e^{-1.5} = 0.3678 - 0.2231 = .1446$$

$$\text{err}_{\max} = e^{-10} - e^{-15} = .000045 - .0000003 = .000045$$

This shows the table should be extended further in the 0.1 range before changing to a delta X of 1.0. However, the accuracy can be controlled to any degree by using more points. As shown in Table 1, 30 points is not a very large table. Three hundred points is still reasonable and could reduce the errors considerably. Use $[0,1,0.01]$, $[1,4,0.03]$, $[4,10,0.06]$, and $[10,100,10]$ as the segments. The maximum errors are

$$\text{err}_{\max} = e^{-0} - e^{-0.005} = 1 - 0.9950 = .00498.$$

$$\text{err}_{\max} = e^{-1} - e^{-1.015} = 0.3676 - 0.3624 = .00547.$$

$$\text{err}_{\max} = e^{-4} - e^{-4.03} = 0.0183 - 0.0177 = .00054.$$

Table 5.3 Segmented Function e^{-x}

x	e^{-x}
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0	
2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0	
20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0	

This is about 5/1000 or 8 bits of accuracy, worst case.

5.5.2 Interpolation

By using segments on the X-axis the previous search to find X_k and X_{k+1} can be used. The computation for $F(x)$ is enhanced by using linear interpolation. Using the previous algorithm, the computation of Y must be changed to:

$$Y = X_F * (Y(X_{k+1}) - Y(X_k)) + Y(X_k). \quad (5.3)$$

In this case a correction is added to compensate for the displacement of X relative to X_k . For the previous case, using the midpoints as examples,

$$X_F = 0.05,$$

$$X_F = 0.5, \text{ and}$$

$$X_F = 5.$$

For these three case,

$$Y_1 = 0.05 Y(0.1) + 0.95 Y(0)$$

$$Y_2 = 0.5 Y(2) + 0.5 Y(1)$$

$$Y_3 = 5 Y(20) - 4 Y(10)$$

Since the exact values are

$$Y_{1ex} = Y(0.05)$$

$$Y_{2ex} = Y(1.5)$$

$$Y_{3ex} = Y(15)$$

the errors in each case are

$$E_1 = Y_{1ex} - Y_1 = Y(0.05) - 0.05Y(0.1) - 0.95Y(0) =$$

$$E_2 = Y_{2ex} - Y_2 = Y(1.5) - 0.5Y(2) - 0.5Y(1) =$$

$$E_3 = Y_{3ex} - Y_3 = Y(15) - 5Y(20) + 4Y(10) =$$

5.6 Parallel Versions of EXOSIM

5.6.1 Version 1.0

The BOOST 2 code was analyzed to determine variable transfers and delayed variables. Figure 5.4 and Table 5.4 are the results. From Figure 5.4, a three processor implementation was constructed. The timing for each processor is roughly indicated by the left-right position in Figure 5.4. This implementation exactly matches the baseline. The baseline executes at 64:1. Figure 5.4 executes at 42:1. The final altitude is 229830.

5.6.2 Version 1.1

The Mass Products block and the ATMOS block within Partition 1 were put on separate processors. These two blocks contain several, one variable look-up tables. These are run in parallel without introducing any variable delays.

The ATMOS subroutine requires ALT for the tables PRESS, RHO, VSND, SHEAR, WINDIR and VWIND. Altitude is output as early as possible from Partition 1 to six processors running these subroutines. The outputs are returned to the Partition 1 processor or to a WINDS processor.

The WINDS processor inputs XYZ, XYZD, and CIM. This enables it to generate LAT and LONG. The answers are equal to the Baseline. Execution time is 32:1.

5.6.3 Version 2.0

A three processor version based on Figure 5.4 was implemented wherein Partitions 1,2 and 3 are all started at $t = 0$ with initial conditions. This implies that Table 5.4 would contain all delayed variables. The execution time was 34:1. The final altitude was 228022.

5.6.4 Version 2.1

The variables on v2.0 were all exchanged at the beginning of the cycle. This causes the transfers to be done in series with the computation. The longest execution time (Partition 1) determines when transfers will start. Execution is started after all transfers are complete.

To introduce parallel operation in computation and transfer, it is necessary to begin processing with initial conditions set for $t=0$. As each partition completes a variable which is to be transferred, a SEND is executed immediately. Variables are grouped for transfer by estimating when they will be ready. This is the most difficult part, since timing information at this level, is not available. However, the timer on each processor can be used to determine the completion time of each of the variables from the start of the cycle until the SEND is executed. Since the Partition division is not complete, this analysis is delayed until later.

The results match those for v2.0. The execution time is _____.

5.6.5 Version 3.0

The errors present in Version 2.0 are due to delays. By estimating these variables it should be possible to reduce the errors and force v2.0 to conform to v1.0. Rather than extrapolate all the variables at once QA, MACH, PRESS and CG were chosen first. These variables were examined using v1.0. A third order extrapolator was used to generate Y_{k+1} from Y_k , Y_{k-1} , Y_{k-2} and Y_{k-3} . When these were tried the execution time became _____:1, with an altitude of _____.

5.6.6 Version 3.1

Figure 5.4 represents the minimal parallel system. By separating partitions 3,4 and 5 and extracting the target subroutine as partition 6, a six processor parallel simulation results as shown in Figure 5.5. The variables which are passed between processors are shown in Figure 5.5 and Table 5.5. All variables passed between partitions are delayed. This caused large errors in altitude and was not considered adequate to pursue.

5.6.7 Version 3.2

Partition 2 is split and placed on two processors. One processor will contain subroutine NCU and BTHRST. The other will contain FRACS and FRCTHST. These will be designated Partition 21 and Partition 22. Partition 3 will be split into three processors. One will contain ACCEL, the second GYRO and the third IMUPRO. These will be designated P31, P32, and P33. Partitons 4 and 5 remain on P33.

5.6.8 Version 4.0

Since double precision variables generate a real problem, it was decided to only transfer the top 32 bits of some double precision variables. Thus PRESS, which is computed and transferred as a 64-bit variable, is still computed as 64-bits, but is transferred as the 32 most significant bits. On the receive end, the variable is input as two 16-bit parts of a double precision number. The other two 16-bit parts are static and never change. (These could be set to zero in the initial program). The result of changing, a variable in this manner is quite minimal. This change leaves one sign bit, eleven exponent bits, one hidden mantissa bit and 20 mantissa bits. The effect of truncating the lower 32-bits introduces an error of less than 10^{-6} . This can be seen by letting V_1 be a 64-bit value and V_2 be the upper 32-bits of V_1 . The error is then defined by,

$$V_1 = 1.M_1 \times 2^E$$

Figure 5.4 Three Processor Implementation - Version 1.0

TABLE 5.4. PARALLEL EXOSIM V1.0 VARIABLE MAP

VARIABLE	PARTITION				
	1	2	3	4	5
MDOTF	DR	S			
MDOTT	DR	S			
FRCX	DR	S			
FRCY	DR	S			
FRCZ	DR	S			
MRCX,Y,Z	DR	S			
FXT, FYT, FZT	DR	S			
MXT, MYT, MZT	DR	S			
P, Q, R	S		R		
UD, VD WD	S		R		
PD, QD, RD	S		R		
CG	S	R	R		
GR	S			R	
ALT	S				R
CGEST	S				R
IT	S				R
VRWM	S				R
SR, SQ		DR		S	
CMMD		DR			S
DLPC, DLYC		DR			S
PM		DR			S
MDLTFR		DR			S
MALPHA		DR			S
QA	S	R			
MACH	S	R			
PRESS	S	R			

$$V_2 = 1.M_2 \times 2^E$$

$$|V_1 - V_2| = (1.M_1 - 1.M_2) \times 2^E$$

$$\leq (1.M_2 + M_3 - 1.M_2) \times 2^E$$

$$\leq (M_3) \times 2^E$$

$$\leq 2^{-20} \times 2^E.$$

The relative error is

$$\frac{|V_1 - V_2|}{|V_1|} \leq \frac{2^{-20}}{1.M_1} \leq 2^{-20}.$$

This accuracy is sufficient for most simulation variables. The exceptions are position coordinates which generate altitude requiring the subtraction of two large numbers. Also, time in EXOSIM is calculated using a double precision technique to ensure accurate alignment of the sample points. A number of if statements contain test constants that require double precision for accurate definition. This timing structure can be converted to single precision, but will require care to examine all the cases which exist. At this point time and all associated variables are being left in "D" format.

The number of variables to be transferred is shown in Table 5.6. This is a three processor implementation of the five partitions. Each partition (processor) requires the number of input/output transfers shown in the table. Note that the total for Partition 1 is 45. If each is transferred as a 64-bit number this will require 180 crossbar cycles for Partition 1 alone. At one microsecond per transfer this will consume 0.18 milliseconds or approximately 20% of the simulation cycle time. There are two ways to reduce this. The first is to transfer 32-bit values instead of 64-bit values. This has the potential to divide the number by 2. A second technique is to divide the partitions and reduce the number of variables for any one processor. _____ variables are now delayed. When this configuration was executed, the time was _____ and the achieved altitude was _____. By implementing prediction on the variables (_____), the altitude was increased to _____, with an increase of execution time to _____.

Figure 5.5 Six Processor Implementation - Version 3.1

TABLE 5.5. PARALLEL EXOSIM v3.0 VARIABLE MAP

VARIABLE	PARTITION					
	1	2	3	4	5	6
MDOTF	DR	S				
MDOTT	DR	S				
FRCX,Y,Z (3)	DR	S				
MRCX,Y,Z (3)	DR	S				
FXT,FYT,FZT (3)	DR	S				
MXT,MYT,MZT (3)	DR	S				
P,Q,R (3)	S		DR			
PD,QD,RD (3)	S		DR			
UD,VD,WD (3)	S		DR			
CG (3)	S	DR	DR	DR		
GR (3)	S				DR	
ALT	S				DR	
CGEST (3)	S				DR	
II (3)	S				DR	
VRWM (3)	S					
MACH	S	DR				
PRESS	S	DR				
QA	S	DR				
CMMD (2)	S	DR			S	
DLPL,DLYC (2)		DR			S	
MALPHA		DR			S	
MDLTFR		DR			S	
PM (3)		DR			S	
SQ,SR (2)		DR		S	DR	
DELPH1,PSI,THT (3)			S	DR		
DELU,V,W			S	DR		
AT (3)				S	DR	
MVR,MVS (2)				S	DR	
TIZM (9)				S	DR	
UVS (3)				S	DR	
RTIC						S
VTIC						S

TABLE5.6. Transfers For Figure 5.4

Partition	TRAN IN	TRAN OUT
1	14	28
2	17	14
3	12	0
4	3	2
5	10	9
Total	59	56

		TO PARTITION					
		1	2	3	4	5	6
FROM PARTITION	1		6	12	3	10	0
	2	14		0	0	0	0
	3	0	0			0	0
	4	0	2	0		19	0
	5	0	9	0	0		0
	6	0	0	0	0	0	

Figure XIII. Processor Transfer Requirements for Figure B.

5-16

5.2
Table XII. Crossbar Schedule for Figure XIII

P A R A L L E L	T R A N S F E R S	1 ~ 2 (6)	4 ~ 5 (6)		
		1 ~ 3 (12)	5 ~ 2 (6)		
		1 ! 4 (3)	5 ~ 2 (3)		
		1 ~ 5 (10)		4 ~ 2 (2) 3 ~ 4 (6)	
		2 ~ 1 (14)	4 ~ 5 (13)		

Figure 5.6 is a matrix of the transfers from each processor to all other processors. This is useful to determine a crossbar transfer pattern. For example row 1 shows all the sends executed by processor 1, whole column 1 shows all the receives executed by processor 1. The total for each processor is important since it sets the absolute minimum number of crossbar cycles. The totals for each processor are: 1:45, 2:31, 3:18, 4:30, and 5:38. Therefore, processor 1 must transfer 45 variables (4, 16-bit words per transfer) and sets the minimum number of transfer cycles. By programming these transfers first, the others can be forced to fit. Therefore schedule 1 to 2 (a), 1 to 3(12), 1 to 4(3), 1 to 5(0), and 2 to 1(14). Since processor 5 has the next highest number of cycles, it is best to try and fit these cycles in next. Table 5.7 shows all the cycles for processor 1 in the first column and those for processor 5 in the second column (with the exception of 1~5). These can be transferred in parallel. The remaining cycles are shown in column 3. This schedule meets the minimum number and is optimal in that sense.

There is another consideration, however. the variables may be computed in an entirely different order. If transfers are to be overlapped with computation for maximum overlap, it is necessary to factor into Table 5.7 timing information which indicates when each of the variables to be transferred by a given processor will be available. Given this information, Table 5.7 can be re-ordered to transfer the variables in the fastest possible manner. This may lead to some transfer conflicts and force extra cycles in the schedule. This information is not being obtained at this point, since more divisions are being sought.

5.6.9 Parallel Versions of Partition 1

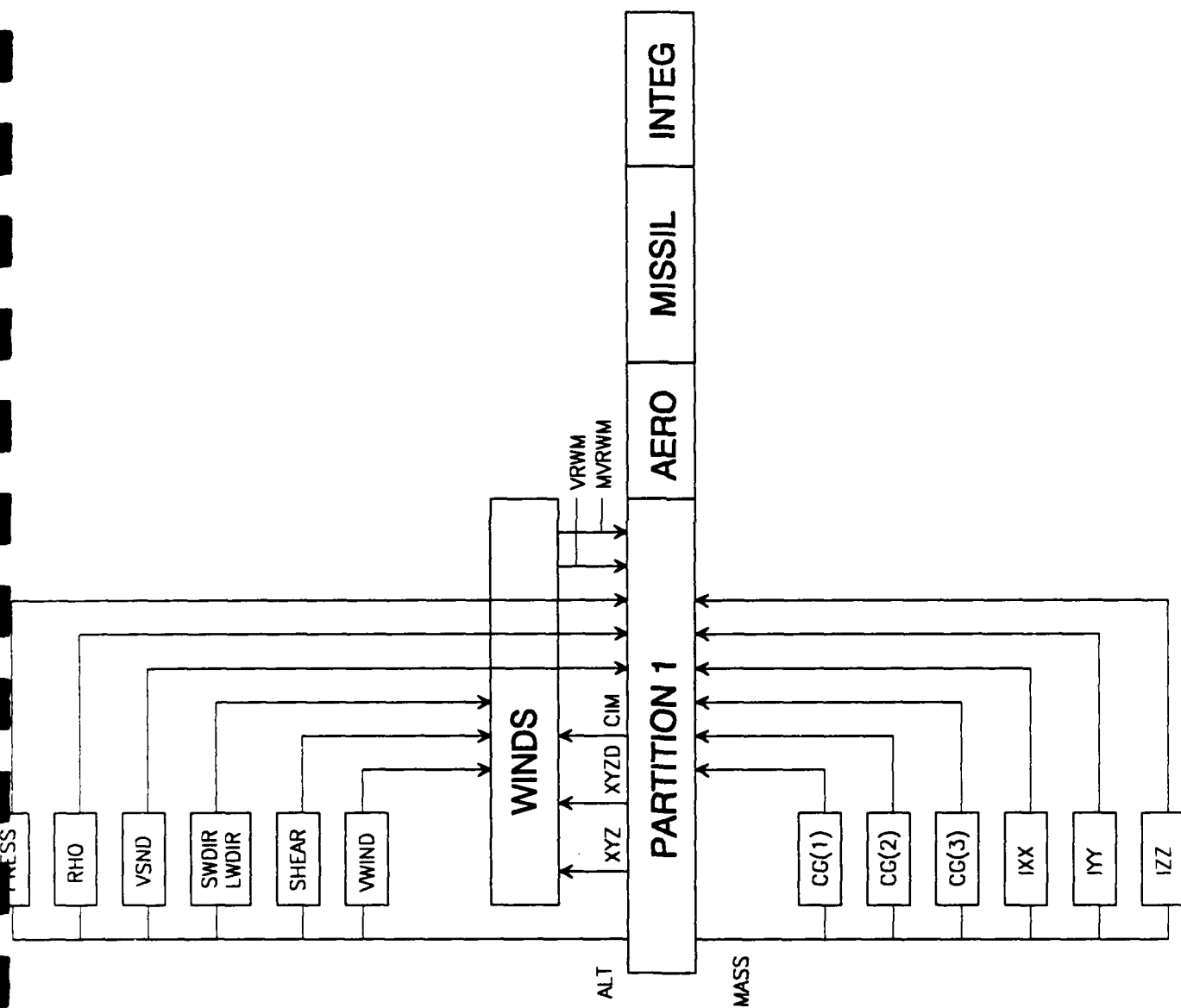
Within Partition 1 are two subroutines which each contain six single variable table look-ups. These tables are driven by ALT for the ATMOS subroutine and by MASS for the MASSPR subroutine. These can be done in parallel with Partition 1 by setting up the tables on individual processors as shown in Figure 5.7. Partition 1 must SEND ALT and MASS to the appropriate processors. Values are returned from the processors to Partition 1 to continue execution.

Since $CG = [CG(1), CG(2), CG(3)]^T = CGEST$, these values can be sent to other partitions at the same time they are sent to Partition 1. The ATMOS subroutine contains six table look-ups and a WINDS subroutine. It is necessary to send XYZ, XYZD and CIM to WINDS in order to calculate Latitude and Longitude. Values from the tables are sent back to WINDS to complete the calculation for VRWM and MVRWM.

Partition 1 can be further divided, as shown in Figure 5.8, by placing AERO on the WINDS processor. Since VRWM and MVRWM only go to the AERO module, this avoids transfers for these two variables. AERO can be divided by placing the four table look-up functions on four processors and directing the results back to the WINDS/AERO processor. After the forces and moments are computed they are passed to MISSIL.

In Figure 5.9, AERO is divided to avoid the transfers MXA, MYA and MZA. These are computed on AERO(2) from the force values transferred from AERO(1). The MISSIL routine in Figure 5.8 has been split into two independent routines. The variables PD, QD, and RD are delayed variables generated by MISSIL (1). These are passed to MISSIL (2) which computes PH1, THT, PSI and CIM. Figure 5.9 represents the state of Partiton 1 as of July 15, 1990. Additional parallelism exists in MISSIL but the most intensive computation now occurs in WINDS. The plan is to convert this to replace the iSBC 286/12 processor executing this code and replace it with an FPP processor. This will require the use of single precision code until the double precision compiler is ready.

6. APPENDIX A



577

Figure A1. Fourteen Processor Version of Partition 1

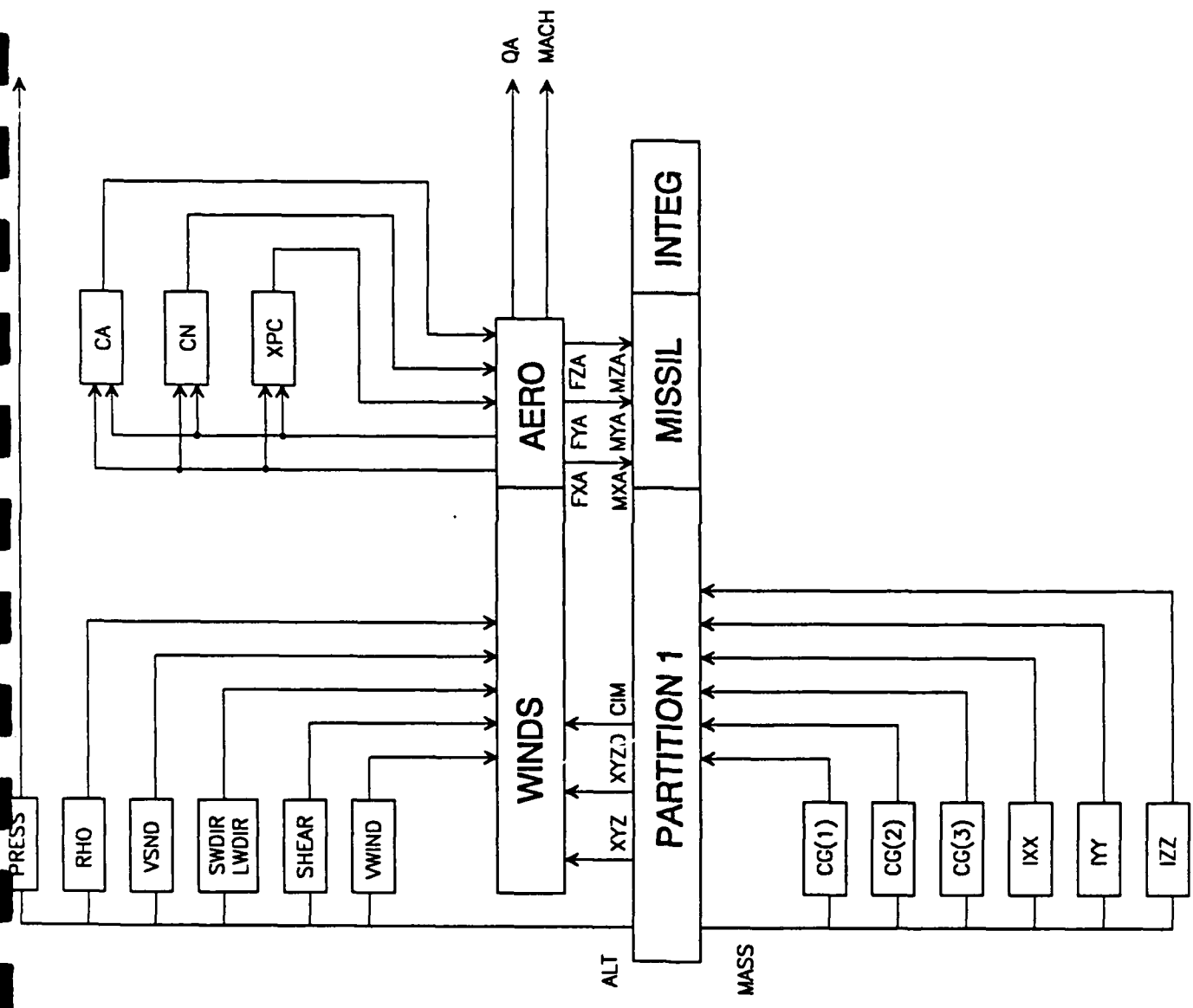


Figure A2. Fourteen Processor Version of Partition 1

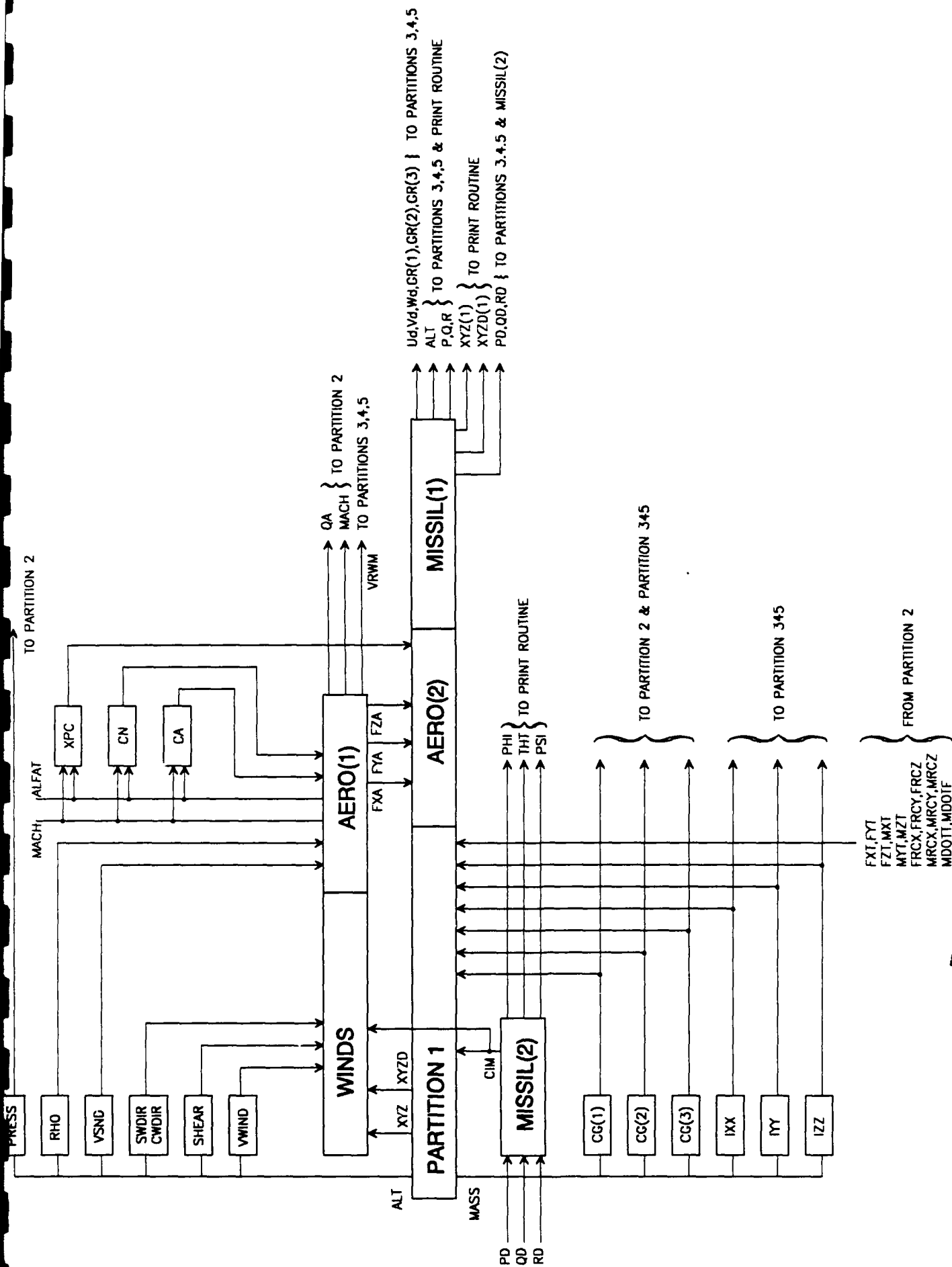


Figure A3. Eighteen Processor Version of Partition 1